Notes on type-driven development with Idris

Christiano Braga

September, 2019

Universidade Federal Fluminense

Contact information

Christiano Braga
Associate Professor

Instituto de Computação
Universidade Federal Fluminense
cbraga@ic.uff.br
http://www.ic.uff.br/~cbraga
Lattes Curriculum Vitae

Introduction

Type-driven development in a nutshell

- Domain-specific languages
  - Focus on what is relevant to the client.
- Program transformation
  - Relates client terminology to the available solutions.
- Structural and behavioral type-safety
  - Allows for both *data* soundness and *process* soundness.
- Transparent use of rigorous program verification techniques.
  - Seamless integration of *mathematically rigorous* techniques into the development process.

Cybersecurity

- Current distributed applications ecosystem: IOT, Cloud, Web...
- A common problem in distributed information systems: *SQL code injection*.
  - Examples: Sony in 2011 and Yahoo! in 2012.
  - Losses of millions of dollars

The problem, by example

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = "
        + txtUserId;
```

If `txtUserId` is equal to `105 OR 1=1`, which is always true, a malicious user may access *all* user information from a database.

Solutions

- SQL parameters: additional values are passed to the query.
- Escaping functions: they transform the input string into a "safe" one before sending it to the DBMS.
- The problem with the solutions is that communication relies on *strings*.
- What if we could **type** this information?

Protocols i

- Web programming invariably requires following certain **protocols**.
  - For example, to connect to make a query:
    1. Create a connection.
    2. Make sure the connection was established.
    3. Prepare an SQL statement.
    4. Execute the query.
    5. Process the result of the query.
    6. Close connection.
- Of course, a function could implement such a sequence, but how could one make sure that such a sequence is *always* followed?

Protocols ii

- In other words, what if we could *type* protocol behavior and make sure our Web programs *cope* with such types?
- Moreover, what if we could define special *notation* to create instances of such types?
- Protocols are one example but note that *business processes* may be treated the same way.

Service-oriented web development model i

Services are blackboxes, are stateless, are composable, among other nice characteristics.

- Services are first-class citizens in Cloud PaaS, and other platforms.
- These characteristics allow for a *clean* and *simple* interpretation of services as *functions*.
- ***What about capturing a company's way of developing PaaS as DSL?***
- ***What about capturing a company's clients processes as DSL?***

An example DSL

(From Fowler&Brady13.)

- Think of each step of a Web application as a business process.
- The notion of a Web application is typed, and so are its steps.
- For example, a Web application has forms and its forms have handlers.
- A particular Web application is *safe* (or well-typed) if its forms are well-typed. A form is well-typed if its handlers are also well-typed.

An example DSL ii

- The database protocol can be captured as a type.

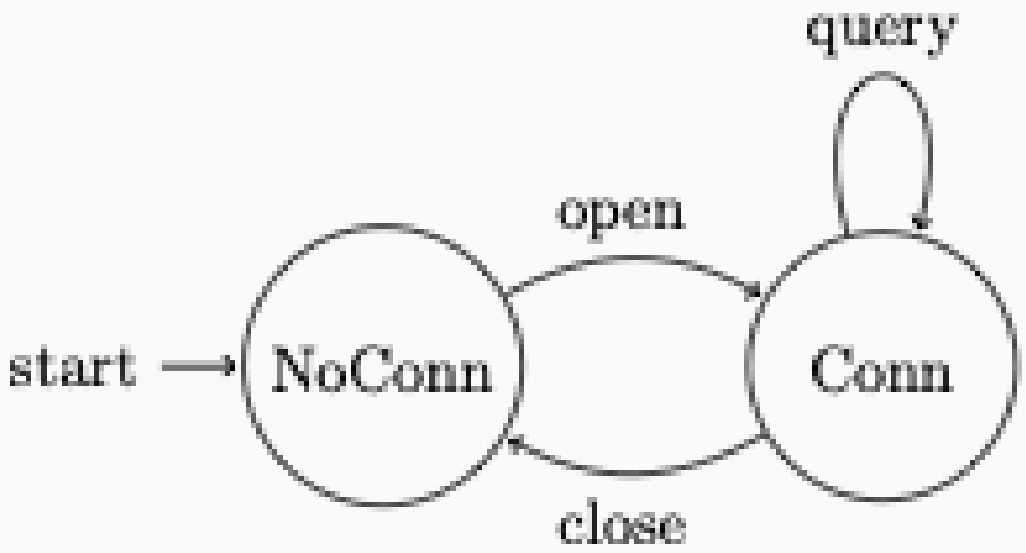

Figure 1: Database protocol

- A program that tries to make a query before opening a connection is **ill-typed**.
  - This is checked at compile time not run time!
  - Your client does not become aware of your errors!

Our research approach

- To program with domain-specific languages, implemented on top of strongly typed functional languages.
- To develop and apply program analysis techniques to DSL-based approaches to software development.

This short-course

- In this short-course we will address some of the basic concepts of the type-driven approach that gives support to the development scenario outlined here.

Suggested reading

Edwin Brady. 2017. Type-driven development. Manning.

Simon Fowler and Edwin Brady. 2013. Dependent Types for Safe and Secure Web Programming. In Proceedings of the 25th symposium on Implementation and Application of Functional Languages (IFL '13). ACM, New York, NY, USA, Pages 49, 12 pages. DOI: https://doi.org/10.1145/2620678.2620683

Golden questions for dependent types

What are dependent types?

- The **future** of programming!
  - Types in programming languages, in a nutshell:
    1. Untyped: assembly languages
    2. Basic native types: floats, integers
    3. Structured types: records, unions
    4. Objects: classes, inheritance, and polymorphism
    5. Generic types: parameterized classes
    6. Algebraic datatypes: inductive types
    7. Generalized algebraic datatypes: parameterized inductive types
    8. **Dependent types: Datatypes that depend on other types and values, Types are first-class citizens.**

Why to develop with dependent types?

- Safer development!
  - Many errors that would otherwise be identified in runtime are captured at compile time. (See my notes on type-driven development)
    - Outstanding for *both* **data** and **behavior** validation.
- Mandatory for *mission critical* development! Examples are:
  - Cyber-security
    - Code injection-free app
  - Cyber-physical systems
    - Type safe hybrid automata
  - Service-oriented development
    - Type safe automata

How to work with dependent types? i

- Idris, programming language developed at St. Andrews Univ., Scotland, UK, lead by Edwin Brady
- Lean, both a theorem prover and programming language developed at Microsoft Research, Redmond, US, lead by Leonardo de Moura
- Coq, theorem prover, is the result of about 30 years of research lead by Thierry Coquand and Gérard Huet, with the contribution of many people.
- F*, programming language developed at Microsoft Research and Inria
- Agda, programming language developed at Chalmers University, Gottenburg, Sweden, lead by Ulf Norell

How to work with dependent types? ii

- Many mainstream programming languages are moving towards dependent type support.

Industrial strength:

- Haskell
  - Haskell in industry site
  - David Thrane Christiansen, Iavor Diatchki, Robert Dockins, Joe Hendrix, Tristan Ravitch, Dependently Typed Haskell in Industry, Proc. ACM Program. Lang., Vol. 3, No. ICFP, Article 100. Publication date: August 2019.
  - Not the full power of dependent types, though

Who works with dependent types?

- In Brazil
  - Academia
    - Yours truly, UFF
    - UFMG, UFPel
- Abroad
  - Academia
    - Inria
    - Microsoft Research
    - Univ. of St. Andrews
  - In industry
    - Coq, Intel Corp., Michael Soegtrop
    - Haskell and Lean, Galois, Inc., Joe Hendrix

The need for dependent types

The need for dependent types

- Overflow conditions in software appear to be a simple thing to implement. An important counter-example is the Ariane 5 rocket that exploded due to a down cast from 64-bit number into a 16-bit one.
  The Ariane 5 had cost nearly $8 billion to develop, and was carrying a $500 million satellite payload when it exploded.
  11 of the most costly software errors in history
- In this chapter we look at a simplified version of the `Vector` datatype, available in Idris' library, to try and understand how *dependent typing* can be useful to have type-safe array handling that could help prevent catastrophes such as the Ariane 5 explosion.

Vector i

- A datatype is nothing but an implementation of some "domain of information". It could very well represent low level information such as data acquired by a sensor in a Internet of Things (IoT) system or the structure that organizes the decision making process in planning.
- Our datatype here is quite simple but illustrates very well how dependent types may help safe data modeling and implementation.

Vector ii

```
> module Vect
> data Vect : Nat -> Type -> Type where
>     Nil  : Vect Z a
>     (::) : (x : a) -> (xs : Vect k a) -> Vect (S k) a
```

- An array or vector is built or *constructed* using either one of the constructor operations (unary) `Nil` or (binary) `::`. (The `module` keyword here simply defines a *namespace* where `Vect` will live.) After loading this file in Idris you could try

```
*tnfdt> 1 :: Vect.Nil
[1] : Vect 1 Integer
```

Vector iii

at the REPL.

- This says that the term `[1]` has type `Vect 1 Integer` meaning that it is a vector with one element and that its elements of the `Integer` type, Idris' basic types.
- Maybe this is a lot to take! *Just breath* and let us think about it for a moment.
- Types are defined in terms of constructor operators. This means that an *instance* of this type is written down as `1 :: Vect.Nil`. In a procedural language you could write it with a code similar to

```
v = insert(1, createVect(1))
```

where `createVect` returns a vector of a given size and `insert` puts an element on the given vector. The point is that we usually create objects or allocate memory to represent data in variables (so called *side effects*) while in functional programming we *symbolically* manipulate them, as in the example above.

- This is a major paradigm-shift for those not familiar with functional programming. Be certain that it will become easier as time goes by, but let's move on!

Dependency i

- Let's look at the instance first and then to the type declaration. Note that the type of `[1]` is `Vect 1 Integer`. The type of a Vect *depends* on its *size*! Think about examples of vectors in programming languages you know. If you query for the type of a given vector, if at all possible, what the run-time of your programming language will answer?
- In Python, for instance, you would get something like,

```
v = [1,2,3]
type(v)
<class 'list'>
```

Dependency ii

that is, is a `list` and that's all! In C an array is a pointer! (A reference to a memory address, for crying out loud!)

- In Idris, we know it is a vector and its size, an important property of this datatype. Cool! And so what?
- We can take advantage of that while programming. We could write a function that does *not*, under no circumstances, goes beyond the limits of a vector, that is, index it beyond its range!

The `zip` function i

- The `zip` function simple creates pairs of elements out of two instances of `Vect` *with the same size*. Here is what it look like:

```
> zip : Vect n a -> Vect n b -> Vect n (a, b)
> zip Nil Nil = Nil
> zip (x :: xs) (y :: ys) = (x, y) :: zip xs ys
```

- What on earth is it? Do you remember how to declare a function in Idris? Well, is pretty-much that. The difference here is that we are now programming with *pattern matching*.
- And what is it? Simply define a function by *cases*.

The `zip` function ii

- When we hit an instance of `Vect`, how does it look like? It is either the empty vector, built with constructor `Nil`, or a non-empty vector, built using operator `::`.
- These two cases are represented by each equation above. The first equation declares the case of "zipping" two *empty* vectors and the second one handles two *non-empty* vectors, specified by the *pattern* `x :: xs`, that is, a vector whose first element is `x` and its remaining elements are represented by a (sub)vector `xs`.
- For instance, if we could write

The `zip` function iii

```
*tnfdt> Vect.zip [1,2,3] ["a", "b", "c"]
[(1, "a"), (2, "b"), (3, "c")] :
     Vect 3 (Integer, String)
```

and get the expected vector of pairs produced by `zip`. (I used `Vect.zip` only because there are other `zip` functions coming from Idris' standard library.)

- Note that the type of `[(1, "a"), (2, "b"), (3, "c")]` is `Vect 3 (Integer, String)` where 3 is the size of the vector and `(Integer, String)`, denoting pairs of integers and strings, is the type of the elements of vector that `zip` calculates.

The `zip` function iv

- Note some additional interesting things about `zip`'s declaration: The signature of `zip` is `zip : Vect n a -> Vect n b -> Vect n (a, b)`. The variable `n` here stands for the size of the vector. Variables `a` and `b` denote the types of the elements of the vectors being zipped.
- That is, the `Vect` type is *generic*, as the type of its elements are underspecified, and is *dependent* on the **number** denoting its size. Again, `n` is a *number*, and `a` (or `b`, for that matter) is a *type*!
- Now, take a look at this:

The `zip` function v

```
*tnfdt> Vect.zip [1,2,3] ["a", "b"]
(input):1:19-21:When checking argument xs to
  constructor Vect.:::
        Type mismatch between
                Vect 0 a (Type of [])
        and
                Vect 1 String (Expected type)

        Specifically:
                Type mismatch between
                        0
                and
                        1
```

The `zip` function vi

- What does this mean? This is a *type checking* error, complaining about an attempt to zip vectors of different sizes. This is *not* an exception, raised while trying to execute `zip`. This is a *compile* type message, regarding the case of zip a vector of length 1 (the last element of the first vector), and a 0-sized vector (from the second vector).

In Idris, types can be manipulated just like any other language construct.

Conclusion.

Ariane 5 would not have exploded (from the bit conversion perspective) if the function that accidentally cast a 64-bit vector into a 16-bit one was written with this approach.

Wrapping-up

1. Defining datatypes.
2. Defining dependent datatypes.
3. Using dependent datatypes to find errors at compile time.
4. Type expressions.

Type-define-refine approach

Type-define-refine approach

- The approach is threefold:

1. Type—Either write a type to begin the process, or inspect the type of a hole to decide how to continue the process.
2. Define—Create the structure of a function definition either by creating an outline of a definition or breaking it down into smaller components.
3. Refine—Improve an existing definition either by filling in a hole or making its type more precise.

Following the TDD book Brady17, we use the Atom editor to illustrate the process. (Idris defines an IDE API such that editors like Atom, Emacs or Vi can interact with the REPL.)

The `allLenghts` function i

- Let us write a function that given a list of strings computes a list of integers denoting the length of each string in the given list.
- Type. Which should be the type for `allLenghts`? Our "problem statement" has already specified it so we just have to write it down:

```
allLenghts : List String -> List Nat
```

- After loading the file `tdr.lidr` we get the following.

The `allLenghts` function ii

```
Type checking ./tdr.lidr
Holes: Main.allLenghts
*tdr> allLenghts
allLenghts : List String -> List Nat
Holes: Main.allLenghts
```

- There is no surprise with the type but there is a `Hole` in our program. Obviously is because we did not declare the equations that define `allLenghts`. This may also occur when Idris fails to type-check a given program.
- Define Idris may help us think about which cases our function must handle. In the Atom editor, we press Ctrl+Alt+A, producing the following definition:

The `allLenghts` function iii

```
allLenghts : List String -> List Nat
allLenghts xs = ?allLenghts_rhs
```

- Of course this is not enough. Here is what Idris says when we load it like this:

```
Type checking ./tdr.lidr
Holes: Main.allLenghts_rhs
```

- Let us think about it: what just happened here? Nothing more than create an equation saying that when the `xs` list is given, "something" `?allLenghts_rhs`-ish will happen. Simple but useful when we repeat this process. It is even more useful as a learning tool. Let's continue!

The `allLenghts` function iv

- Idris won't leave us with our hands hanging here. It can assist us on thinking about what `?allLenghts_rhs` should look like if we inspect `xs`.
- If we press Ctrl+Alt+C on `xs` the editor spits out the following code:

```
allLenghts : List String -> List Nat
allLenghts [] = ?allLenghts_rhs_1
allLenghts (x :: xs) = ?allLenghts_rhs_2
```

The `allLenghts` function v

- Two equations were produced because lists in Idris are defined either as the empty list, denoted by `[]`, or a non-empty list denoted by the *pattern* `x :: as`, where `x` is the first element of the given list, which is concatenated to the rest of list in `xs` by the operator `::`.
- Nice, and now we have two holes to think about, when the given list is empty and otherwise. Idris allows us to check the type of each hole using the command Ctrl+Alt+T when the cursor is on top of each variable.

The `allLenghts` function vi

```
--------------------------------------
allLenghts_rhs_1 : List Nat

x : String
xs : List String
--------------------------------------
allLenghts_rhs_2 : List Nat
```

- Refine. The refinement of `allLenghts_rhs_1` is trivial: Ctrl+Alt+S (*proof search*) on it gives us `[]`.

The `allLenghts` function vii

- For `allLenghts_rhs_2` we need to know however that there exists a `length` operation on strings. We should than apply it `x` and "magically" build the rest of the resulting string. Our code now looks like this:

```
allLenghts : List String -> List Nat
allLenghts [] = []
allLenghts (x :: xs) = (length x) :: ?magic
```

- Atom and Idris may help us identify what kind of magic is this. We just have to Ctrl+Alt+T it to get:

The `allLenghts` function viii

```
x : String
xs : List String
-----------------------------------
magic : List Nat
```

- So now we need *faith on recursion* (as Roberto Ierusalimschy, a co-author of Lua, says) and let the rest of the problem "solve itself". Finally, we reach the following implementation:

```
> module Main
>
> allLenghts : List String -> List Nat
```

The `allLenghts` function ix

```
> allLenghts [] = []
> allLenghts (x :: xs) = (length x) :: allLenghts xs
```

- Awesome! For our final magic trick, I would like to know if Idris has a function that given a string produces a list of strings whose elements are the substrings of the first. Try this on the REPL:

```
*type-define-refine/tdr> :search String -> List String
= Prelude.Strings.lines : String -> List String
Splits a string into a list of newline separated strings.
= Prelude.Strings.words : String -> List String
Splits a string into a list of whitespace separated
```

The `allLenghts` function x

`strings.`

`...`

- It turns out that `words` is exactly what I was looking for! Run the following:

```
*type-define-refine/tdr>
:let l = "Here we are, born to be kings,
                we are princess of the universe!"
*type-define-refine/tdr> words l
["Here",
 "we",
 "are,",
 "born",
```

The `allLenghts` function xi

```
"to",
"be",
"kings,",
"we",
"are",
"princess",
"of",
"the",
"universe!"] : List String
```

- And Finally

The allLenghts function xii

```
*type-define-refine/tdr> :let w = words l
*type-define-refine/tdr> allLenghts w
[4, 2, 4, 4, 2, 2, 6, 2, 3, 8, 2, 3, 9] : List Nat
```

Lab i

In the labs in this short-course you will have to complete or fix some Idris code.

- First lab.

The first lab is to complete the code below using what we have discussed so far.

```
> wordCount : String -> Nat
> -- Type-define-refine this function!
> -- Start by running `Ctrl+Alt+A` to add a definition,
> -- than `Ctrl+Alt+C` to split cases and finally
> -- `Ctrl+Alt+S` to search for proofs(!) that represent
```

Lab ii

```
> -- the code you need! (Intrigued? Ask the instructor
> -- for an advanced course on this topic than = )
>
> average : (str : String) -> Double
> average str =
>         let numWords = wordCount str
>             totalLength =
>                   sum (allLenghts (words str))
>         in ?w
> -- Which is the type of `?w1`?
> -- Proof search won't help you here, unfortunately...
> -- Run `:doc sum` at the REPL. Just read the
> -- documentation at the moment, not the type of `sum`.
```

Lab iii

```
>
> showAverage : String -> String
> showAverage str =
>   let m = "The average word length is: "
>       a = average ?w
>   in m ++ show (a) ++ "\n"
> -- Check the type o `w` and think about it!
>
> main : IO ()
> main = repl "Enter a string: " showAverage
```

- Using the example string from above, you should get the following spit at you:

Lab iv

```
Sat Aug 03@18:05:17:type-define-refine$
idris --nobanner tdr.lidr
Type checking ./tdr.lidr
*tdr> :exec main
Enter a string:
Here we are, born to be kings,
  we are princess of the universe!
The average word length is: 3.923076923076923
```

- Moreover, you may *compile it* to an executable with the following command line:

```
idris --nobanner tdr.lidr -o tdr
```

Lab v

and then execute it, as follows.

```
Sun Aug 04@12:39:21:type-define-refine$ ./tdr
Enter a string:
```

Insertion sort lab.

Insertion sort lab.

- Here is what we will implement:
- Given an empty vector, return an empty vector.
- Given the head and tail of a vector, *sort* the tail of the vector and then insert the head into the sorted tail such that the result remains sorted.
- At the end, you should be able to run the following at the REPL:

```
*VecSort> insSort [1,3,2,9,7,6,4,5,8]
[1, 2, 3, 4, 5, 6, 7, 8, 9] : Vect 9 Integer
```

I will first walk you through the development of most of the code. At the end of the section I list your activities for this lab.

Type-define-refine i

- Type We will use the `Vect` datatype available in Idris' prelude.

```
> import Data.Vect
```

And it is easy to grasp the signature of our function, so here it goes.

```
insSort : Vect n elem -> Vect n elem
```

- Define Now we add a clause using `Ctrl+Alt+A` on `inSort`, resulting in

```
insSort : Vect n elem -> Vect n elem
insSort xs = ?insSort_rhs
```

and do a case split on variable `xs`.

Type-define-refine ii

```
insSort : Vect n elem -> Vect n elem
insSort [] = ?insSort_rhs_1
insSort (x :: xs) = ?insSort_rhs_2
```

- Refine

```
insSort : Vect n elem -> Vect n elem
insSort [] = []
insSort (x :: xs) = ?insSort_rhs_2
```

- Proof search works just fine for `?insSort_rhs_1` but not so much for `?insSort_rhs_2`, as it simply produces

```
insSort (x :: xs) = ?insSort_rhs_2
```

Type-define-refine iii

- And why is that? Because there is no *silver bullet* and you need to understand the algorithm! The informal specification is quite clear: we need to insert `x` into a sorted (tail) list.

```
insSort (x :: xs) =
    let l = insSort xs in ?insSort_rhs_2
```

- We can now ask the system to help us with `?insSort_rhs_2` in this context by pressing Ctrl+Alt+L on it. Here is what it creates:

Type-define-refine iv

```
insSort_rhs_2 : (x : elem) -> (xs : Vect len elem) ->
                (l : Vect len elem) -> Vect (S len) elem
insSort (x :: xs) =
  let l = insSort xs
  in (insSort_rhs_2 x xs l)
```

It generates a *stub* of a function with all the variables in the context.

- Since we are following quite easily =(what is going on, we now that we need to rename `insSort_rhs_2` to `insert` (just for readability) and get rid of `xs` in the application, leaving us with

```
insSort (x :: xs) = let l = insSort xs in (insert x l)
```

Type-define-refine v

- Awesome! Let us now define `insert` as the lifting process (with Ctrl+Alt+L) already (overly)defined its type for us. So let us add a clause on `insert`, and case-split `l`. It leaves us with the following code once we search for a proof for hole 1.

```
insert : (x : elem) -> (l : Vect len elem)
                    -> Vect (S len) elem
insert x [] = [x]
insert x (y :: xs) = ?insSort_rhs_2
insSort : Vect n elem -> Vect n elem
insSort [] = []
insSort (x :: xs) = let l = insSort xs in (insert x l)
```

Type-define-refine vi

- Proof search will not help us with hole 2, as there are some things we need to figure out. Let us think for a moment what `insert` should do. There are two cases to consider:
- If x $<$ y, the result should be x :: y :: xs, because the result won't be *ordered* if x is inserted after y.
- Otherwise, the result should begin with y, and then have x inserted into the tail xs.
- In a *type safe* context we need to make sure that `insert` will be able to compare `x` and `y`. In object-oriented terms, that object `x` knows how to answer to message < or that the algebra of `x` and `y` is an order!

Type-define-refine vii

- Idris implements the concept of *type classes*, called `interfaces` in Idris and are precisely that: they define operations that a certain datatype must fulfill.
- One such type class is `Ord`.

```
interface Eq a => Ord a where
    compare : a -> a -> Ordering

    (<) : a -> a -> Bool
    (>) : a -> a -> Bool
    (<=) : a -> a -> Bool
    (>=) : a -> a -> Bool
```

Type-define-refine viii

```
max : a -> a -> a
min : a -> a -> a
```

- It relies on yet another type class called `Eq`, that defines the equality relation and defines a number of operations, including <. Type-classes form an important concept in strongly-typed functional programming but we will not explore it any further in this short-course.
- Having said that, we need to constraint `insert` such that `elem` is an *ordered* type.

Type-define-refine ix

```
> insert : Ord elem => (x : elem) ->
>                      (l : Vect len elem) -> Vect (S len) elem
> insert x [] = [x]
> insert x (y :: xs) = ?insert_rhs

> insSort : Ord elem => Vect n elem -> Vect n elem
> insSort [] = []
> insSort (x :: xs) = let l = insSort xs in (insert x l)
```

Lab activities

- So, finally, here is what you should do:

1. Perform all the steps described above until you reach the code above.
2. Replace the meta-variable with the appropriate `if then else` code or search for Ctrl+Alt+M (to generate a `case`-based code) command on the web and try i.t

Programming with type-level functions

Programming with type-level functions

- Here are a couple of examples where first-class types can be useful:
  - Given an HTML form on a web page, you can calculate the type of a function to process inputs in the form.
  - Given a database schema, you can calculate types for queries on that database. In other words, the type of a value returned by a database query may vary *depending on* the database *schema* and the *query* itself, calculated by **type-level functions**.
- This should be useful in a number of contexts such as Data validation in Robotic Process Automation, SQL Injection, (Business) Process Protocol Validation, just to name a few.
- In this section we discuss and illustrate how this way of programming is available in the Idris language.

Formatted output example i

- This examples explores some of the components for the RPA scenario. It exemplifies how to make strings from properly-typed data using type-functions, similarly to the `printf` function in the C programming language.

```
> module Format
>
> data Format =
> Number Format
> | Str Format
> | Lit String Format
> | End
```

Formatted output example ii

- The `Format` datatype is an *inductive* one: is a "list" such that its elements are either `Number`, `Str`, `Lit s` (where `s` is string) or `End`. It will be used to *encode*, or to represent, in Idris, a formatting string.
- Try this at the REPL:

```
*pwfct> Str (Lit " = " (Number End))
Str (Lit " = " (Number End)) : Format
```

- This instance of `Format` represents the formatting string "%s = %d" in C's `printf`.
- So far, nothing new, despite the fact that we now realize that our datatypes can be recursive.

Formatted output example iii

- Function `PrintfType` is a *type-level function*. It describes the *functional type* associated with a format.

```
> PrintfType : Format -> Type
> PrintfType (Number fmt) = (i : Int) -> PrintfType fmt
> PrintfType (Str fmt) = (str : String) -> PrintfType fmt
> PrintfType (Lit str fmt) = PrintfType fmt
> PrintfType End = String
```

- Recall that a functional type is built using the `->` constructor. The first equation declares that a `Number` format is denoted by an `Int` in the associated type. The remaining equations define similar denotations.

Formatted output example iv

- Try this at the REPL:

```
*pwfct> PrintfType (Str (Lit " = " (Number End)))
String -> Int -> String : Type
```

- As I mentioned before, the format `(Str (Lit " = " (Number End)))` encodes the C formatting string "%s = %d". The functional type that denotes it is `String -> Int -> String`, that is, a function that receives a string and an integer and returns a string.

Formatted output example v

- Again, `PrintfType` is a type-function, that is, it defines a type. Of course, we can use it to specify, for instance, the return type of a function. The recursive function `printfFmt` receives a format, a string and returns a term of `PrintfType` that *depends on the format given as first argument*!

```
> printfFmt : (fmt : Format) ->
>             (acc : String) -> PrintfType fmt
> printfFmt (Number fmt) acc =
>           \i => printfFmt fmt (acc ++ show i)
> printfFmt (Str fmt) acc =
>           \str => printfFmt fmt (acc ++ str)
```

Formatted output example vi

```
> printfFmt (Lit lit fmt) acc =
>           printfFmt fmt (acc ++ lit)
> printfFmt End acc = acc
```

- Function `toFormat` is a normal function that transforms a string denoting a format and creates a *type* `Format`. Function `printf` is defined next.

```
> toFormat : (xs : List Char) -> Format
> toFormat [] = End
> toFormat ('%' :: 'd' :: chars) = Number (toFormat chars)
> toFormat ('%' :: 's' :: chars) = Str (toFormat chars)
> toFormat ('%' :: chars) = Lit "%" (toFormat chars)
```

Formatted output example vii

```
> toFormat (c :: chars) =
>   case toFormat chars of
>     Lit lit chars' => Lit (strCons c lit) chars'
>     fmt => Lit (strCons c "") fmt
> printf : (fmt : String) ->
>          PrintfType (toFormat (unpack fmt))
> printf fmt = printfFmt _ ""
```

- Try this out at the REPL:

Formatted output example viii

```
*pwfct> :let msg =
        "The author of %s, published in %d, is %s."
*pwfct> :let b = "A Brief History of Time"
*pwfct> :let a = "Stephen Hawking"
*pwfct> :let y = the Int 1988
*pwfct> printf msg b y a
"The author of A Brief History of Time,
 published in 1988, is Stephen Hawking." : String
```

- At this point you should be able =(to understand what is going on. Why does `printf` takes four arguments? Shouldn't it be just one? (The `fmt : String` above.)

Formatted output example ix

- For variable `y` we had to make sure it is an `Int` (finite), not an `Integer` (infinite) number, due to `PrintfType` definition. This is what `the Int 1988` does. Try it without the casting and see what happens...

Conclusion

- The point here is that we can use types to help organize the world.
- Recall the SQL Injection example from the introductory section. The problem there was the fact that everything was a string.
- Using the concepts discussed here we could type information coming from forms and check them before sending them to the DBMS!

Caveats i

(From TDD book.)

- In general, it's best to consider type-level functions in exactly the same way as ordinary functions. This isn't always the case, though. There are a couple of technical differences that are useful to know about:
- Type-level functions exist at *compile* time only. There's no runtime representation of Type, and no way to inspect a Type directly, such as pattern matching.

Caveats ii

- Only functions that are total will be evaluated at the type level. A function that isn't total may not terminate, or may not cover all possible inputs. Therefore, to ensure that type-checking itself terminates, functions that are not total are treated as constants at the type level, and don't evaluate further.

Infinite data and processes

Infinite data i

- Streams are infinite sequences of values, and you can process one value at a time.
- When you write a function to generate a Stream, you give a prefix of the Stream and generate the remainder recursively. You can think of an interactive program as being a program that produces a potentially infinite sequence of interactive actions.

Infinite data ii

```
> %default total
> data InfIO : Type where
>      Do : IO a -> (a -> Inf InfIO) -> InfIO
> (>>=) : IO a -> (a -> Inf InfIO) -> InfIO
> (>>=) = Do
> loopPrint : String -> InfIO
> loopPrint msg = do putStrLn msg
>                    loopPrint msg
> partial
> run : InfIO -> IO ()
> run (Do action cont) = do res <- action
>                           run (cont res)
```

Infinite data iii

- Try the following at the REPL:

```
:exec run (loopPrint "on and on and on...")
```

and a non-terminating execution will present itself. As expected, `run` is *not* total:

```
*streams/streams> :total run
Main.run is possibly not total due to recursive path:
    Main.run, Main.run
```

- The type `InfIO`, as the name suggests, is a type of infinite IO actions, denoted by the type variable `a`. The `Do` constructor receives an IO action and produces an infinite IO action, by recursion.

Infinite data iv

- Function `loopPrint` is one such *action generator*.
- Let us take this slowly: First of all, what is the `Inf` type?

```
Inf : Type -> Type
Delay : (value : ty) -> Inf ty
Force : (computation : Inf ty) -> ty
```

- `Inf` is a generic type of potentially infinite computations.
- `Delay` is a function that states that its argument should only be evaluated when its result is forced.
- Force is a function that returns the result from a delayed computation.

Another example with infinite data i

- `InfList` is similar to the List generic type, with two significant differences:
  - There's no `Nil` constructor, only a `(::)` constructor, so there's no way to end the list.
  - The recursive argument is wrapped inside `Inf`.

```
> data InfList : Type -> Type where
>     (::) : (value : elem) -> Inf (InfList elem) ->
>     InfList elem
```

- Function `countFrom` is an example on how to use `Inf`.

Another example with infinite data ii

```
> countFrom : Integer -> InfList Integer
> countFrom x = x :: Delay (countFrom (x + 1))
```

The Delay means that the remainder of the list will only be calculated when explicitly requested using Force.

Try the following at the REPL:

```
*streams> countFrom 0
0 :: Delay (countFrom 1) : InfList Integer
```

Streams i

- Idris has streams in its prelude.

```
data Stream : Type -> Type where
    (::) : (value : elem) -> Inf (Stream elem) ->
                                 Stream elem
repeat : elem -> Stream elem
take : (n : Nat) -> (xs : Stream elem) -> List elem
iterate : (f : elem -> elem) -> (x : elem) -> Stream elem
```

- Execute

Streams ii

```
 (iterate (+1) 0)
*streams/streams> (iterate (+1) 0)
0 ::
Delay (iterate (\ARG => prim__addBigInt ARG 1) 1) :
                                        Stream Integer
```

and try to grasp which type is this.

- Here are some cool stuff we can do with streams, try it out:

```
Idris> take 10 [1..]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] : List Integer
```

The syntax [1..] generates a Stream counting upwards from 1.

Streams iii

- This works for any countable numeric type, as in the following example:

```
Idris> the (List Int) take 10 [1..]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] : List Int
```

or

```
Idris> the (List Int) (take 10 [1,3..])
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19] : List Int
```

- Now, which is the relationship between all this machinery and the motivation presented at the beginning of the course?
  - Are there any relations among IOT sensors and streams?

Streams iv

- You should probably have realized by now that `run` is an *infinite process* executing on an *infinite stream* of data!

Making infinite processes total i

- As trivial as it may sound, a way to make a function terminate is simply to define a "time out".
- In the following example, this is denoted by the `Fuel` datatype. The `Lazy` datatype is similar to the `Inf` we have seen before, it "encapsulates" infinite data and only computes it when necessary.

```
> data Fuel =
>  Dry | More (Lazy Fuel)
>
> tank : Nat -> Fuel
```

Making infinite processes total ii

```
> tank Z = Dry
> tank (S k) = More (tank k)
>
> partial
> runPartial : InfIO -> IO ()
> runPartial (Do action f) =
>            do res <- action
>               runPartial (f res)
>
> run2 : Fuel -> InfIO -> IO ()
> run2 (More fuel) (Do c f) =
>      do res <- c
>         run2 fuel (f res)
```

Making infinite processes total iii

```
> run2 Dry p = putStrLn "Out of fuel"
>
> partial
> main : IO ()
> main = run2 (tank 10) (loopPrint "vroom")
```

`Inf` vs. `Lazy`

- If the argument has type Lazy ty, for some type ty, it's considered smaller than the constructor expression.
- If the argument has type Inf ty, for some type ty, it's not considered smaller than the constructor expression, because it may continue expanding indefi- nitely. Instead, Idris will check that the overall expression is productive

Protocols

A trivial database protocol

- The automaton below illustrates the communication between an application and a database system. The intention is to express that in order to query a database it is necessary first to establish a connection with it and then after all queries were done, the connection is closed.

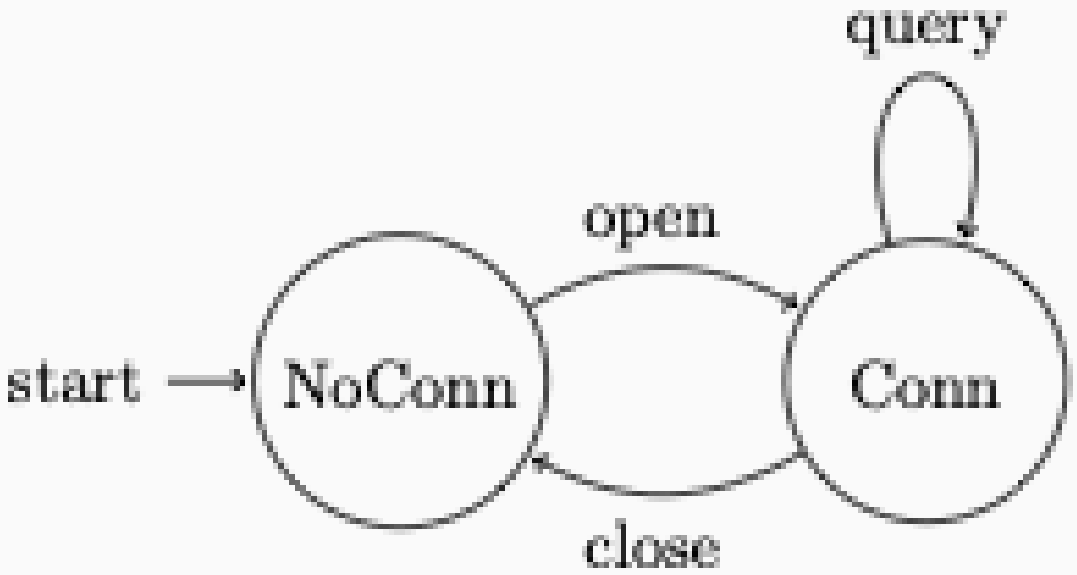

Figure 2: Trivial database protocol

First attempt: a monoid of actions i

- The code below is a naïve implementation of it.

```
> module DBProtocol
>
> import Data.Vect
>
>
> data DBConnState = Conn | NotConn
>
> namespace DBCmd1
>
>     data DBCmd : Type -> Type where
```

First attempt: a monoid of actions ii

```
>           Open : DBCmd ()
>           Close : DBCmd ()
>           Query : DBCmd ()

>           Pure : ty -> DBCmd ty
>           (>>=) : DBCmd a -> (a -> DBCmd b) -> DBCmd b
```

- Program `dbProg1` does exactly that.

```
>     dbProg1 : DBCmd ()
>     dbProg1 = do Open
>                  Query
>                  Close
```

First attempt: a monoid of actions iii

- But `dbProg2` also type checks just fine. Think about it for a moment. *Why is this the case?*

```
>     dbProg2 : DBCmd ()
>     dbProg2 = do Close
>                  Open
>                  Query
```

- Transitions are not *typed*! We can combine them in any way we want, in the code. But this is not the "spirit" of the specification. (Mathematically speaking, we do *not* want a *free* monoid of actions but rather an *ordered* one!)

Second attempt: a partial order i

- We can do better and we will. We can type transitions by annotating, each operation in `DBCmd` type, with the *source* and *target* types.
- This is captured in the type with signature

```
data DBCmd : Type -> DBConnState -> DBConnState -> Type.
```

- On each transition, for instance in `Open`, with the following signature:

```
Open : DBCmd () NotConn Conn
```

where `DBCmd ()` is its (returning) type.

Second attempt: a partial order ii

- Types `NotConn` and `Conn` are the types of the source and target states that specify, respectively, the pre and postconditions of the `Open` action.

```
> namespace DBCmd2
>
>     data DBCmd : Type -> DBConnState ->
>                          DBConnState -> Type where
>          Open : DBCmd () NotConn Conn
>          Close : DBCmd () Conn NotConn
>          Query : DBCmd () Conn Conn
>
```

Second attempt: a partial order iii

```
>          Pure : ty -> DBCmd ty state state
>          (>>=) : DBCmd a state1 state2 ->
>                  (a -> DBCmd b state2 state3) ->
>                  DBCmd b state1 state3
>
>     dbProg1 : DBCmd () NotConn NotConn
>     dbProg1 = do Open
>                  Query
>                  Close
```

- The sequence of actions `Open`, `Query` and `Close` types correctly, as expected.

Second attempt: a partial order iv

```
dbProg2 : DBCmd () NotConn NotConn
dbProg2 = do Query
             Close
             Open
```

- However, if a program tries to query a database to which there is no open connection, the program simply does not type-check!
- We can check it simply using command

```
idris --check protocol.lidr
```

Second attempt: a partial order v

as, in this example, there are not implementations for `Query`, `Close` and `Open`.

```
Tue Aug 13@16:06:57:protocols$
idris --check protocol.lidr
protocol.lidr:89:20-24:
   |
89 | >      dbProg2 = do Query
   |                     ~~~~~
When checking right hand side of
DBProtocol.DBCmd2.dbProg2
  with expected type
        DBCmd () NotConn NotConn
```

Second attempt: a partial order vi

```
When checking an application of constructor
DBProtocol.DBCmd2.>>=:
        Type mismatch between
                DBCmd () Conn Conn (Type of Query)
        and
                DBCmd a NotConn state2 (Expected type)

        Specifically:
                Type mismatch between
                        Conn
                and
                        NotConn
```

A simple app

Introduction i

- In this section we build on top of the implementation of the `Database` protocol we have just created.
- Before, we were interested, essentially, on specifying a datatype that captures the *behavior* (or automaton) of the protocol, guaranteeing that a computation (or transition) takes place only when its contract (pre and postconditions) hold.
  - In other words, we specify when *computations* are *well-formed*.
- Now we wish to write a *running* application on top of it. It has a command-line interface and uses a table or map (`SortedMap`, in Idris) to represent a database.

Putting it all together i

- Our app requires a few extensions with respect to what we have done so far:
  1. A way to transform string input into "commands" (a well-formed instance of a datatype.)
  2. A way to represent the database.
  3. A way to evaluate commands in the presence of a database.
  4. An updated protocol datatype that takes into account queries and reports.
  5. An interactive user-interface.
- You should note that we are essentially putting everything we studied together.

A way to transform string input into “commands” i

- Our app will simply open a database, close a database and query it. So let us first define a datatype that captures these three commands, and call it `Input`.

```
data Input = OPEN String
           | CLOSE
           | QUERY QueryLang
```

- A query should not be defined as a string, as always, we should type it! Of course, we will not define SQL here but focus on three commands:
  - `INSERT` adds an entry to the database composed by an `Integer` and a `String`.

A way to transform string input into “commands” ii

- `SELECT` retrieves the String bound to the given integer.
- `DELETE` removes from the database the entry whose key is the given integer.

- The `QueryLang` datatype implements it.

```
data QueryLang = INSERT Int String
               | SELECT Int
               | DELETE Int
```

- We now define the transformation function from Strings to the `Input` datatype.

```
strToInput : String -> Maybe Input
```

A way to transform string input into “commands” iii

An example application of this function is:
`strInput "query INSERT 1 A"` $\rightsquigarrow$ `QUERY INSERT 1 "A"`.

- Essentially, it should handle three classes of strings, one for each form of input. Note also that both `OPEN` and `QUERY` have *parameters*.
- Think about this function and try to figure it out by yourself!
- You may check my proposed solution later in the slides.
- I suggest two auxiliary functions: `mkQuery` and `parseQuery`.
  - Function `mkQuery : String -> Maybe (Input)` decomposes the input string and calls `parseQuery` to build the `Input` instance.

A way to transform string input into “commands” iv

- Function `parseQuery : List String -> Maybe Input` receives a list of strings, such as `["query", "INSERT 1 A"]`, and produces an instance of the `Input` datatype, such as `QUERY INSERT 1 "A"`.

A way to represent the database i

- We chose to represent the database as a map (`SortedMap`), available in the Idris distribution.

```
Database : Type
Database =  SortedMap Int String
```

- The type `Database` is imply a synonym to a map from integers to strings.
  - Of course we could relate richer structures with the map and even create a more realistic representation of a database.
  - This simple map should suffice given our pedagogical needs at this time.
- To use it we need to:

A way to represent the database ii

  - Import it in our program with: `import Data.SortedMap`
  - and invoke Idris using the command line: `idris -p contrib simple-app.lidr`

- This will inform the run-time that we are importing the `SortedMap` and where to find it (in package `contrib`).

A way to evaluate commands in the presence of a database i

- We define function by structural induction (the constructors `INSERT`, `SELECT` and `DELETE`) of the datatype `QueryLang` and relate each constructor with an operation of datatype `SortedMap`.
    - Of course, a more realistic implementation wouldn't define a simple bijection (one-to-one), but, again, enough for our pedagogical needs.
- This about it! You may *cheat* and look the proposal solution if you will. But think hard first!

A way to evaluate commands in the presence of a database ii

```
eval : QueryLang -> Database -> (Database, Report)
eval (INSERT i s) db = ?i
eval (SELECT i) db = ?s
eval (DELETE i) db = ?d
```

- Use the command `:browse Data.SortedMap` to learn about `SortedMap`'s interface.
- The notation `(Database, Report)` simply defines a *pair* of `Database` and `Report` where the latter is simply a list of strings.

An updated protocol datatype i

- As before, we have transitions to open, close and query the database.
- However, we now have a more refined notion of *state* of the database app (`DBState`) comprised by the name of open database, its connection status, the database itself and the report of the last query.
- We must update the type of the datatype and of its transitions.
- As always, think about it and cheat if you feel like it...
- A sorted map may be initialized with the `fromList` command. (Search for it in Idris' REPL.)

An updated protocol datatype ii

```
data DBCmd : Type -> DBState -> DBState -> Type
where
  OPENDB : (d : String) ->
   DBCmd () (s, NotConn, db, [])
            (..., ..., ..., ...)
  CLOSEDB :
   DBCmd () (s, Conn, db, r)
            (..., ..., ..., ...)
  QUERYDB : (q : QueryLang) ->
   DBCmd () (s, Conn, db, r)
     (s, Conn, fst (...), snd (...))
  Display : String -> DBCmd () st st
```

An updated protocol datatype iii

```
GetInput : DBCmd (Maybe Input) st st
Pure : ty -> DBCmd ty state state
(>>=) : DBCmd a state1 state2 ->
        (a -> DBCmd b state2 state3) ->
        DBCmd b state1 state3
```

An interactive user-interface i

- Streams are the way to go to write app with infinite data.
- This is exactly what happens when we write interactive applications.
- The datatype `DBIO` defines an stream of instances of `DBState`. Note the use of the `Inf` constructor, while defining a trace of `DBState` with the `Do` constructor...

```
data DBIO : DBState -> Type where
     Do : DBCmd a state1 state2 ->
          (a -> Inf (DBIO state2)) -> DBIO state1
```

- ... which is precisely what we need to implement the lifting of `(>>=)` to sequences of `DBCmd`.

An interactive user-interface ii

- Now we need to define a function that will interact with the user and enact the appropriate actions given a well-formed input. Function dbLoop does precisely that. Again, it is defined by cases on the possible states.
- Understand the following implementation and think about the missing cases captured by the ellipsis.

```
dbLoop : DBIO st
dbLoop {st = (n, NotConn, d, [])} =
 do Just x <- GetInput
            | Nothing =>
               do Display "Invalid input"
                  dbLoop
```

An interactive user-interface iii

```
      case x of
       ...
       otherwise =>
        do Display
             "You should open the database first."
           dbLoop

dbLoop {st = (n, Conn, d, r)} =
 do Just x <- GetInput
            | Nothing =>
               do Display "Invalid input"
                  dbLoop
    case x of
```

An interactive user-interface iv

```
CLOSE =>
 do CLOSEDB {s = n} {db = d}
    dbLoop
...
otherwise =>
 do Display
      "Either close or query the database."
    dbLoop
```

- Function dbLoop executes sequences of commands. We need to be able to "connect" it with the IO system of Idris' run time.
- From the user's perspective, dbLoop must be ran "forever". And that is precisely what main does.

An interactive user-interface v

```
main : IO ()
main =
 run forever
      (dbLoop
        {st = ("", NotConn, fromList [(0,"0")], [])})
```

- Function `run` makes the connection I mentioned above, relating `DBIO` instances with `IO` instances.

```
run : Fuel -> DBIO state -> IO ()
run (More fuel) (Do c f)
   = do res <- runMachine c
        run fuel (f res)
run Dry p = pure ()
```

An interactive user-interface vi

- Datatype `DBIO` is a sequence of DB commands. Function `run` only "iterates" over the infinite sequence of commands, processing it step-by-step by means of function `runMachine`.
- And it does it using the *lazy* datatype `Fuel` (that we studied before), that allows `run` to execute DB commands one step at the time, with a `DBIO` (infinite) sequence.
- Let us take a look at the `runMachine` function. It is defined by cases on `DBCmd` datatype. We will only study one of its cases. The remaining ones are for you think about.

An interactive user-interface vii

```
runMachine : DBCmd ty inState outState -> IO ty
runMachine
 {inState = (s, NotConn, db, [])}
 {outState = (s', Conn, (fromList [(0, "0")]), [])}
 (OPENDB s') =
 do
  putStrLn ("DB " ++ s' ++ " open")
  showDB (fromList [(0, "0")])
```

An interactive user-interface viii

- Function `runMachine` relates a DB command and IO actions. In the case of command `OPENDB s`, where `s` is a string, denoting the name of the database, `runMachine` prints that the database, whose name was given, is open and lists the contents of an initialized database.

Simple app full listing

Datatypes i

```
> import Data.SortedMap
>
> namespace Database
>
>     data ConnState = NotConn | Conn
>
>     Report : Type
>     Report = List String
>
>     Database : Type
>     Database =  SortedMap Int String
```

Datatypes ii

```
>
>     DBState : Type
>     DBState = (String, ConnState, Database, Report)
>
>     data QueryLang = INSERT Int String
>                    | SELECT Int
>                    | DELETE Int
>
>     data Input = OPEN String
>                | CLOSE
>                | QUERY QueryLang
```

Function `mkInsert i`

```
>     mkInsert : List String -> Maybe QueryLang
>     mkInsert xs =
>      case tail' xs of
>       Just y =>
>        case y of
>          s1 :: [s2]  => Just (INSERT (cast s1) s2)
>          otherwise => Nothing
>       otherwise => Nothing
```

Function `mkSelect i`

```
>     mkSelect : List String -> Maybe QueryLang
>     mkSelect xs =
>      case tail' xs of
>       Just y =>
>        case y of
>         [s] => Just (SELECT (cast s))
>         otherwise => Nothing
>       otherwise => Nothing
```

Function `mkDelete i`

```
>     mkDelete : List String -> Maybe QueryLang
>     mkDelete xs =
>      case tail' xs of
>       Just y => case y of
>          [s] => Just (DELETE (cast s))
>          otherwise => Nothing
>       otherwise => Nothing
```

Function parseQuery i

```
>     parseQuery : List String -> Maybe Input
>     parseQuery xs =
>      case head' xs of
>       Just "INSERT" =>
>        case mkInsert(xs) of
>         Just q => Just (QUERY q)
>         Nothing => Nothing
>       Just "SELECT" =>
>        case mkSelect(xs) of
>         Just q => Just (QUERY q)
>         Nothing => Nothing
```

Function `parseQuery` ii

```
>         Just "DELETE" =>
>          case mkDelete(xs) of
>            Just q => Just (QUERY q)
>            Nothing => Nothing
>         otherwise => Nothing
```

Function mkQuery i

```
>      mkQuery : String -> Maybe (Input)
>      mkQuery "" = Nothing
>      mkQuery  s =
>       let h = head' (words s)
>       in
>        case h of
>         Just "query" =>
>          let xs = tail' (words s)
>          in case xs of
>               Just y => parseQuery(y)
```

Function `mkQuery` ii

```
>                otherwise => Nothing
>            otherwise => Nothing
```

Function strToInput i

```
>     strToInput : String -> Maybe Input
>     strToInput s =
>                 if ((head' (words s)) == (Just "open"))
>                 then
>                   let db = tail' (words s)
>                   in
>                       case db of
>                         Just d =>
>                           case d of
>                             [s'] => Just (OPEN s')
>                             otherwise => Nothing
```

Function strToInput ii

```
>                           otherwise => Nothing
>                 else
>                      if s == "close"
>                      then Just CLOSE
>                      else mkQuery(s)
```

Function `eval i`

```
>     eval : QueryLang -> Database -> (Database, Report)
>     eval (INSERT i s) db = ((insert i s db), [])
>     eval (SELECT i) db =
>         case lookup i db of
>             Just s => (db , [s])
>             otherwise => (db, [])
>     eval (DELETE i) db = ((delete i db), [])
```

Datatype DBCmd i

```
>     data DBCmd : Type -> DBState -> DBState -> Type
>     where
>       OPENDB : (d : String) ->
>        DBCmd () (s, NotConn, db, [])
>                 (d, Conn, (fromList [(0,"0")]), [])
>       CLOSEDB :
>        DBCmd () (s, Conn, db, r)
>                 ("", NotConn, db, [])
>       QUERYDB : (q : QueryLang) ->
>        DBCmd () (s, Conn, db, r)
>          (s, Conn, fst (eval q db), snd (eval q db))
```

Datatype DBCmd ii

```
>         Display : String -> DBCmd () st st
>         GetInput : DBCmd (Maybe Input) st st
>         Pure : ty -> DBCmd ty state state
>         (>>=) : DBCmd a state1 state2 ->
>                 (a -> DBCmd b state2 state3) ->
>                 DBCmd b state1 state3
```

Datatype DBIO i

```
>     data DBIO : DBState -> Type where
>          Do : DBCmd a state1 state2 ->
>               (a -> Inf (DBIO state2)) -> DBIO state1
```

Function `showDB i`

```
>       showDB : Database -> IO ()
>       showDB db =
>        if null db
>        then putStrLn ""
>        else
>         putStrLn (show (zip (keys db) (values db)))
```

Function `runMachine i`

```
>       runMachine : DBCmd ty inState outState -> IO ty
>       runMachine
>        {inState = (s, NotConn, db, [])}
>        {outState = (s', Conn, (fromList [(0, "0")]), [])}
>        (OPENDB s') =
>        do
>         putStrLn ("DB " ++ s' ++ " open")
>         showDB (fromList [(0, "0")])
```

Function `runMachine` ii

```
>      runMachine
>       {inState = (s, Conn, db, r)}
>       {outState = ("", NotConn, db, [])}
>       CLOSEDB = putStrLn ("DB " ++ s ++ " closed")
```

Function runMachine iii

```
>     runMachine
>      {inState = (s, Conn, db, r)}
>      {outState = (s, Conn,
>        (fst (eval q db)),
>        (snd (eval q db)))}
>      (QUERYDB q) =
>       do putStrLn("DB contents")
>          showDB   (fst (eval q db))
>          putStrLn("Query result")
>          putStrLn (unwords (snd (eval q db)))
```

Function `runMachine` iv

```
>     runMachine (Pure x) = pure x
>     runMachine (cmd >>= prog) = do x <- runMachine cmd
>                                    runMachine (prog x)
>     runMachine (Display str) = putStrLn str
>     runMachine {inState = (s, c, db, r)} GetInput
>       = do putStr ("DB: " ++ s ++ "> ")
>            x <- getLine
>            pure (strToInput x)
```

Fuel, forever and run i

```
>     data Fuel = Dry | More (Lazy Fuel)
>
>     partial
>     forever : Fuel
>     forever = More forever
>
>     run : Fuel -> DBIO state -> IO ()
>     run (More fuel) (Do c f)
>       = do res <- runMachine c
>            run fuel (f res)
>     run Dry p = pure ()
```

Function >>= lifted to streams of DBCmd i

```
>      namespace DBDo
>        (>>=) : DBCmd a state1 state2 ->
>                (a -> Inf (DBIO state2)) -> DBIO state1
>        (>>=) = Do
```

Function dbLoop i

```
>     dbLoop : DBIO st
>     dbLoop {st = (n, NotConn, d, [])} =
>      do Just x <- GetInput
>                 | Nothing =>
>                    do Display "Invalid input"
>                       dbLoop
>         case x of
>          OPEN x =>
>           do OPENDB x {db = d}
>              dbLoop
>          otherwise =>
```

Function dbLoop ii

```
>             do Display
>                  "You should open the database first."
>                dbLoop
>
>     dbLoop {st = (n, Conn, d, r)} =
>      do Just x <- GetInput
>                 | Nothing =>
>                    do Display "Invalid input"
>                       dbLoop
>         case x of
>          CLOSE =>
>           do CLOSEDB {s = n} {db = d}
>              dbLoop
```

Function dbLoop iii

```
>            (QUERY q) =>
>             do QUERYDB q {s = n} {db = d}
>                dbLoop
>            otherwise =>
>             do Display
>                 "Either close or query the database."
>                dbLoop
```

Function `main` i

```
> main : IO ()
> main =
>  run forever
>        (dbLoop {st = ("", NotConn,
>                       fromList [(0,"0")], [])})
```